Kozio® VTOS™
Evaluation & Tutorial Guide

Evaluation Guide

KOZIO®

Kozio, Inc.
2015 Ionosphere Street Suite 201
Longmont, CO 80504 USA
Office: 303.776.1356

Contents

1 Introduction

Welcome to Kozio VTOS – the software designed to provide design verification and test for embedded designs. VTOS is a special purpose Verification and Test Operating System (VTOS) that first comes into play when the first prototype (simulated or real) is ready and then provides a production test solution eliminating staff months or years of test software development. VTOS can immediately verify new designs and isolate design errors at the earliest stages so that hardware bugs do not go undetected. Fully automated manufacturing test capability is also available for certifying manufactured boards before shipment and thus reducing support costs. The software is delivered ready-to-run with an extensive library of test primitives and test suites, avoiding internal development time for diagnostics.

1.1 About this tutorial

This tutorial is designed to introduce you to VTOS by providing you with a series of interactive lessons that guide you through many of the software's key features. Each lesson is organized around a project that illustrates particular aspects of VTOS.

Each lesson is self-contained, so you may follow the lessons in whatever order best suits your needs and interests. However, if you are new to VTOS, you should familiarize yourself with the VTOS environment by completing Lesson 1: The Basics first. Although the lessons take you on a planned tour of the software, you are of course encouraged to vary the lessons and experiment with VTOS in whatever way satisfies your curiosity.

1.2 Evaluation platform information

VTOS supports a wide range of evaluation platforms, and consequently, this tutorial may refer to features that are not supported on your evaluation platform. An effort has been made to select examples and features that are common to most evaluation platforms, while still illustrating the use of important VTOS features in depth. You are encouraged to read through lessons that may not exactly fit your evaluation platform, as you can still learn important VTOS operating principles from these lessons. Kozio is adding new test suites and tutorial lessons with each release, so please drop us a line requesting the information you need to make an informed decision.

1.3 Prerequisites

Before using this tutorial, you should have the VTOS software running on your evaluation platform.

1.4 Additional resources

This VTOS tutorial is not intended to be a comprehensive manual for the VTOS software. The tutorial only describes what is necessary to understand and complete the lessons. Consult the following additional resources for more information about VTOS:

- ***VTOS Command Reference***. This reference manual contains a complete description of the test suites, diagnostic tests, test methods, and low-level commands. Available upon request.

- ***Integration Workbench User Manual***. This reference manual contains information regarding the operation of Integration Workbench which is used and the console interacting with VTOS.

2 Introduction to Integration Workbench

Integration Workbench (IW) is the user console for interacting with VTOS.

2.1 Quick Tour

Integration Workbench is a powerful user interface built for the purpose of design verification. IW has the ability to run commands via a Command Line or user configurable Command Tree interface. All output generated by VTOS is displayed in the VTOS Output Window. After applying power to the device under test (DUT), boot messages will be displayed. Once the boot sequence is complete the DUT online indicator will display “DUT online” and the command line will become active.

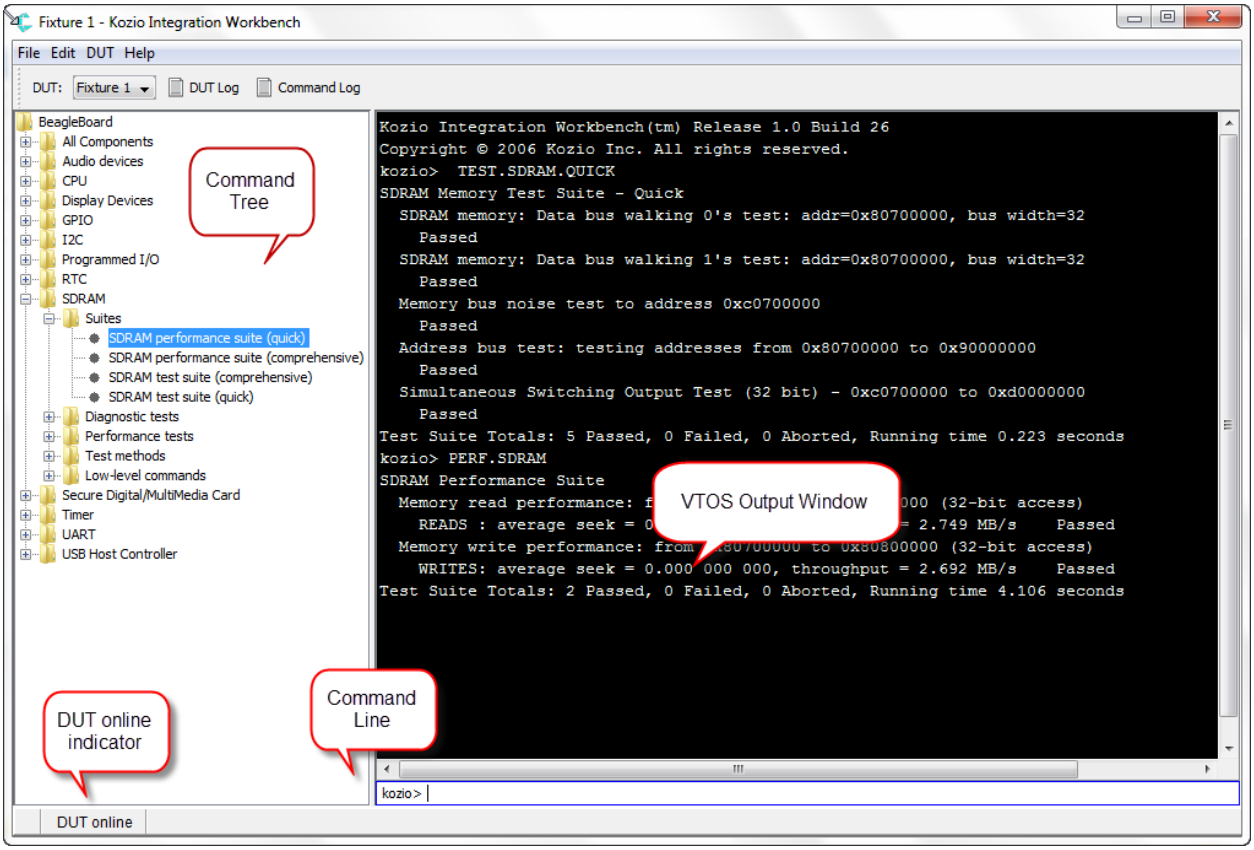


Figure 2.1: Quick Tour

2.2 Loading the Command Tree

A platform specific command tree can be loaded in the left window pane to provide a list of the most commonly used commands for easy execution. The organization of the tree is described in the next section. To load your platform specific Command Tree:

- Within Integration Workbench select “File/Load Command Tree”
- Select the appropriate command tree – E.g. CommandTree-BeagleBoardXM.kct or CommandTree-Blaze.kct.

2.3 Command Tree Hierarchy

Each command tree component is organized into one to five sections: Low-Level Commands, Test Methods, Performance Tests, Diagnostic Tests, and Test Suites.

2.3.1 Low-Level Commands

Low-Level Commands provide the most direct access to the hardware. Many low-level commands are available for the operator to execute, each requiring specific input parameters. Each high-level component, such as SDRAM, may have low-level commands associated with them.

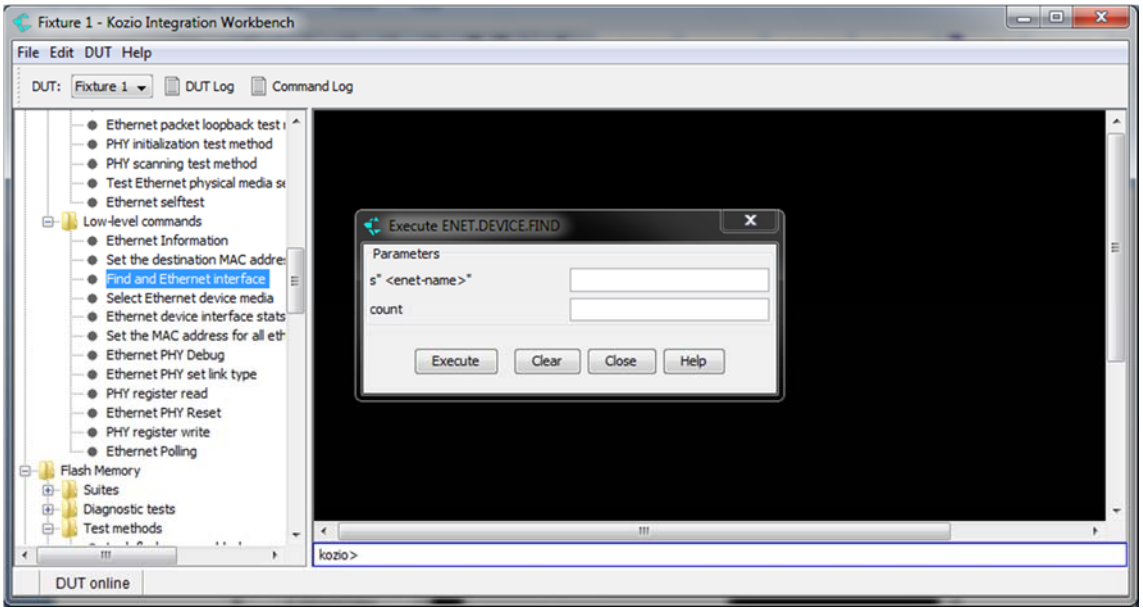


Figure 2.2: Low-Level Commands

2.3.2 Test Methods

Test methods are a set of diagnostic tests that deliver a pass/fail result. Most test methods require input parameters.

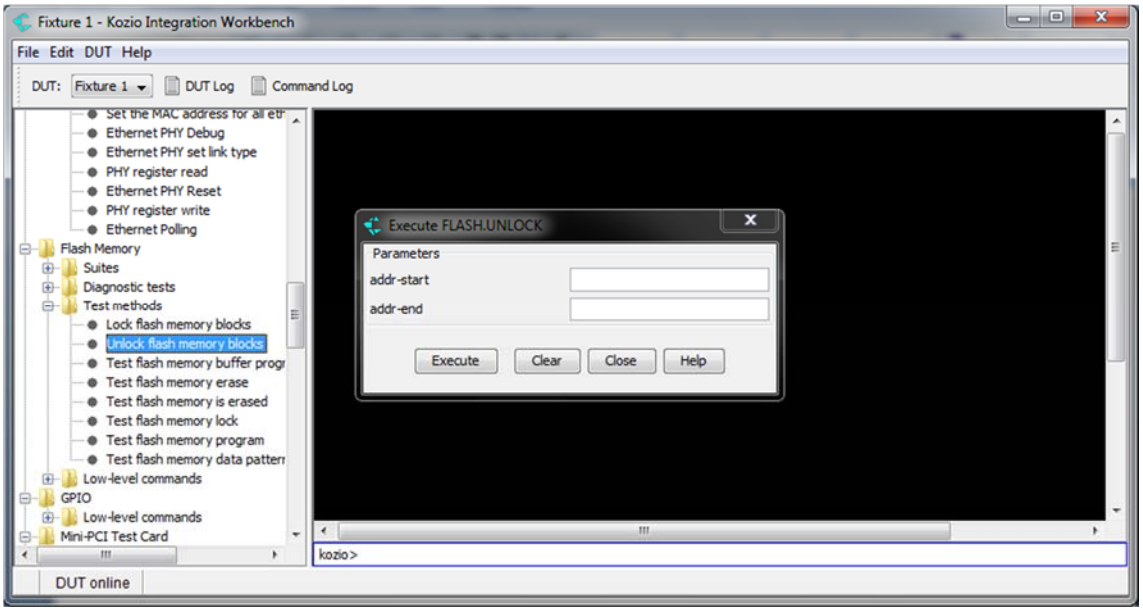


Figure 2.3: Test Methods

2.3.3 Performance Tests

Some components, such as SDRAM, have a number of performance tests associated with them. The Output Window displays a characterization of a given performance characteristic.

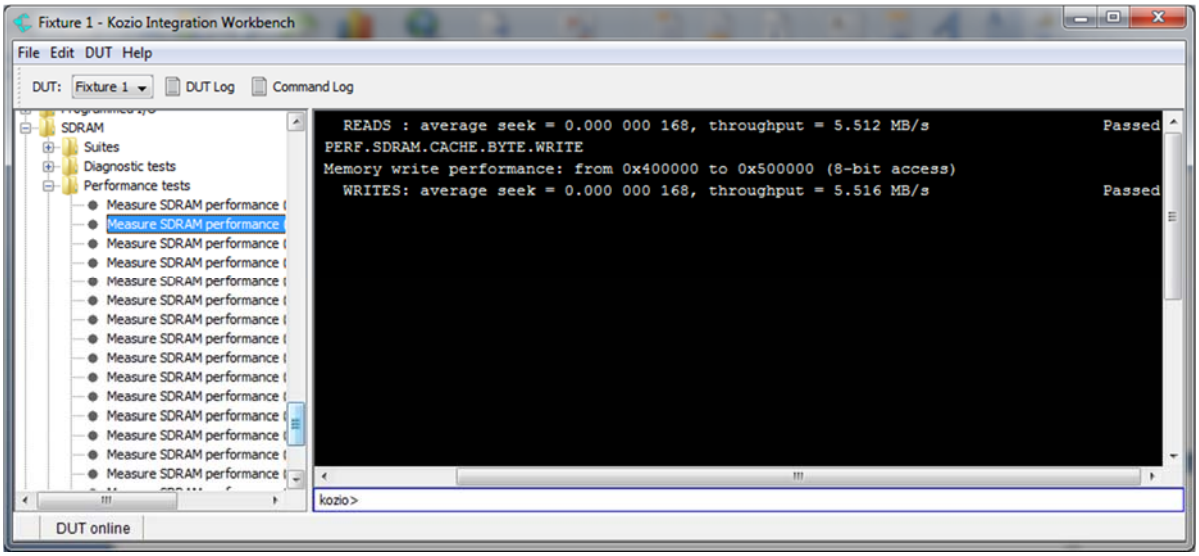


Figure 2.4: Performance Tests

2.3.4 Diagnostic Tests

A set of diagnostic tests has been built on top of test methods using default input parameters. The tests are pre-configured to your platform based on the layout and build for your design. The diagnostic tests deliver a pass/fail result and do not require input parameters from the user.

2.3.5 Test Suites

Test suites are collections of diagnostic tests and/or other test suites that are executed in sequence. Quick and comprehensive test suites are available for all devices. Test suites can also be nested. For example, under the ***All Components*** folder in the Command Tree, over 140 tests are packaged into a single command labeled ***Example Test Suite – long version***. Multiple components are tested and there are multiple levels of nesting under each suite.

In addition to the right-click menu options of ***Help*** and ***Execute***, the Test Suites also have two additional menu options: ***Tree*** and ***Report***. Both commands are accessed by right-clicking on a highlighted test suite and selecting the command from the pop-up menu. The ***Tree*** and ***Report*** commands will display a complete listing of all of the nested suites, and tests included in the suite, in the Output Window. The ***Report*** will also include the current test status: Pending, Pass, or Fail. This same information can be accessed by entering the "**tree**" or "**report**" command followed by the test suite name on the Command Line. For example, on the Command Line enter "**report test.sdram**" (without the quotes) and press ***Enter***.

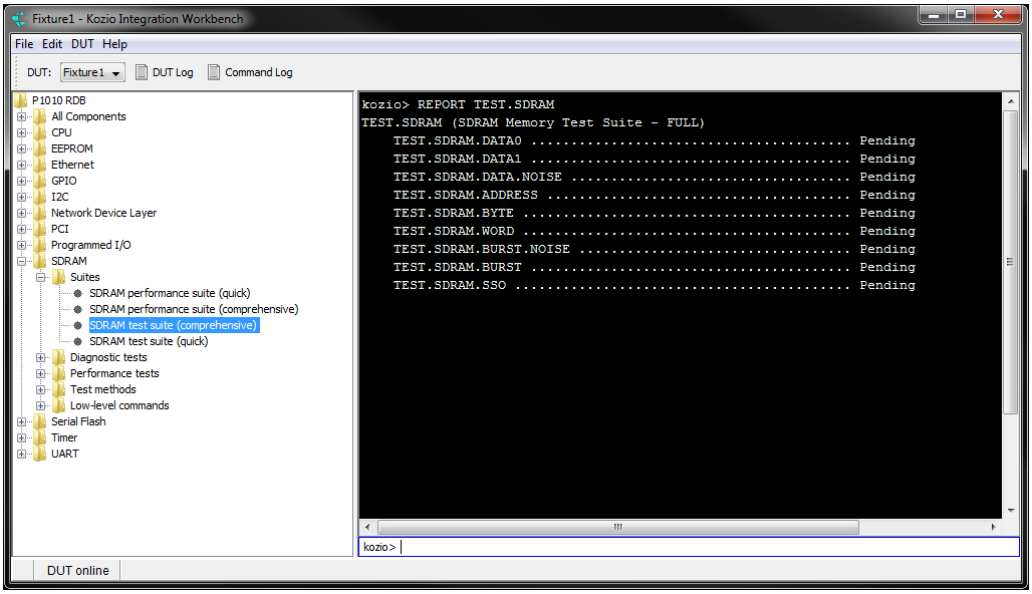


Figure 2.5: Test Suites Report Output

2.4 Using the Command Tree

Commands in the command tree are executed by double clicking the requested test. Users can expose additional options by right clicking on the desired command. All entries provide ***Help*** and ***Execute*** options; Test Suites also provide ***Tree*** and ***Report***.

- ***Execute*** – Run the selected command
- ***Report*** – Generate a results report for the last test run of the current session.
- ***Tree*** – The tree option will list the tests that are part of the selected command.
- ***Help*** – A brief explanation of the test contents

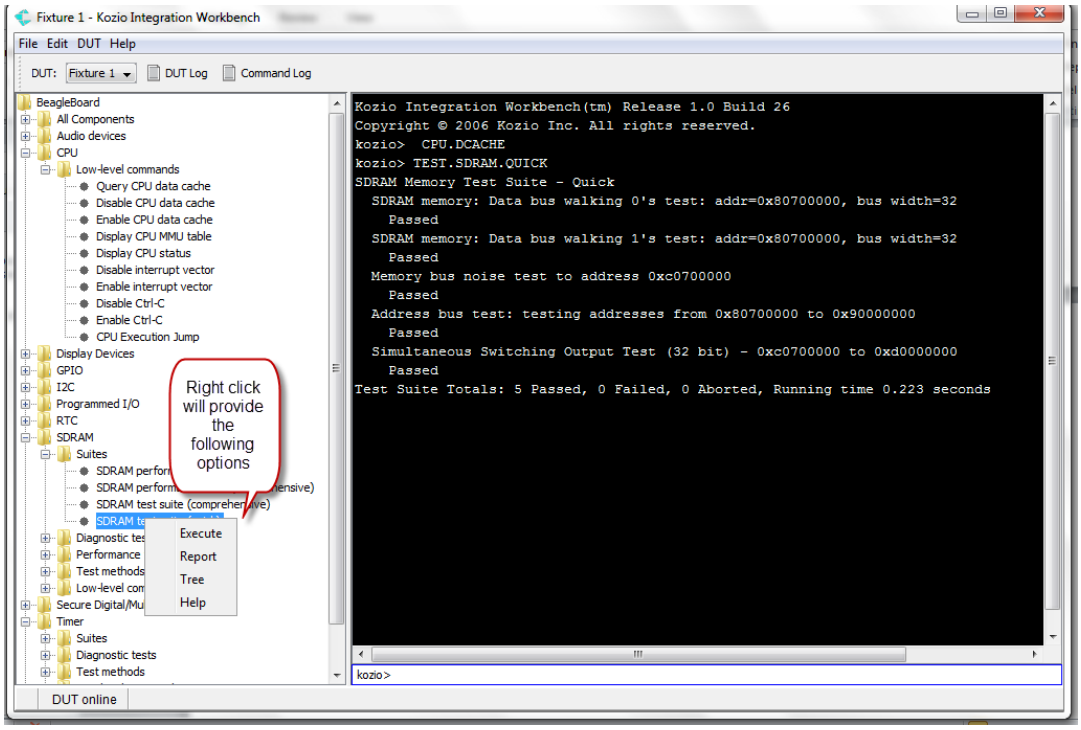


Figure 2.6: Right-Clicking in the Command Tree

2.5 DUT Output Capture and Command Log

Integration Workbench offers the user a command log for recording issued commands. Select the "Command Log" button and a dialogue will open requesting the name of a log file. The user can create multiple log files to capture different types of test or debug activities. This may be useful when developing custom scripts. The files are stored as standard text and can be opened later for review with a text editor such as Notepad.

Users can also save DUT output in a log file similar to the command log.

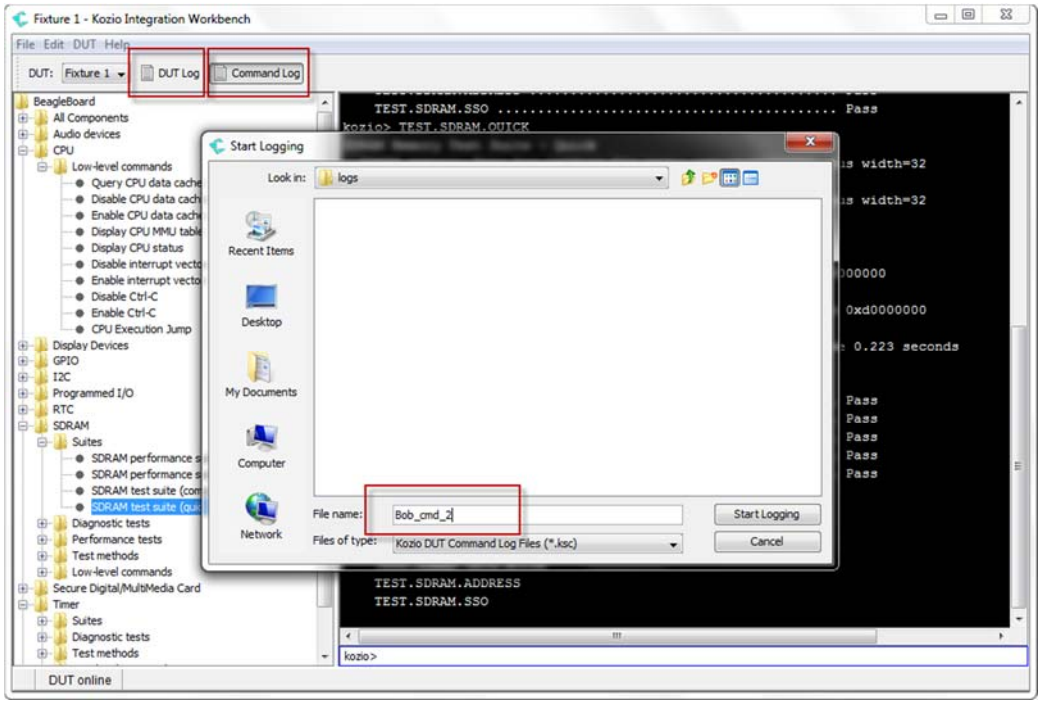


Figure 2.7: DUT Log

3 Basic User Interface Introduction

In this lesson, you will learn to use the most basic features of the VTOS command interpreter using a command line interface to interact with the evaluation platform.

3.1 About this lesson

In this lesson, you will learn how to do the following:

- Launch VTOS.
- Run VTOS commands by entering them at the command prompt.
- Interrupt VTOS command execution.
- Use the VTOS command history and command editing facilities.
- Automate VTOS commands using script files.

This lesson will take approximately 60 minutes to complete.

3.2 Launching VTOS

Consult the appropriate ***evaluation application note*** for your specific platform for instructions on how to connect the evaluation platform to your host computer and properly configure Integration Workbench on your host to communicate with the target.

VTOS will launch automatically whenever the evaluation platform is powered up.

1. Start Integration Workbench on your host computer.
2. Power up the evaluation platform.

When the boot sequence is complete, the Command Prompt below the Output Window will become active, and **DUT online** is displayed in the status bar. VTOS is now ready to accept your commands.

3.3 Issuing commands

VTOS indicates that it is connected and ready to accept a new command by displaying “DUT online” and enabling the “kozio>” command prompt.

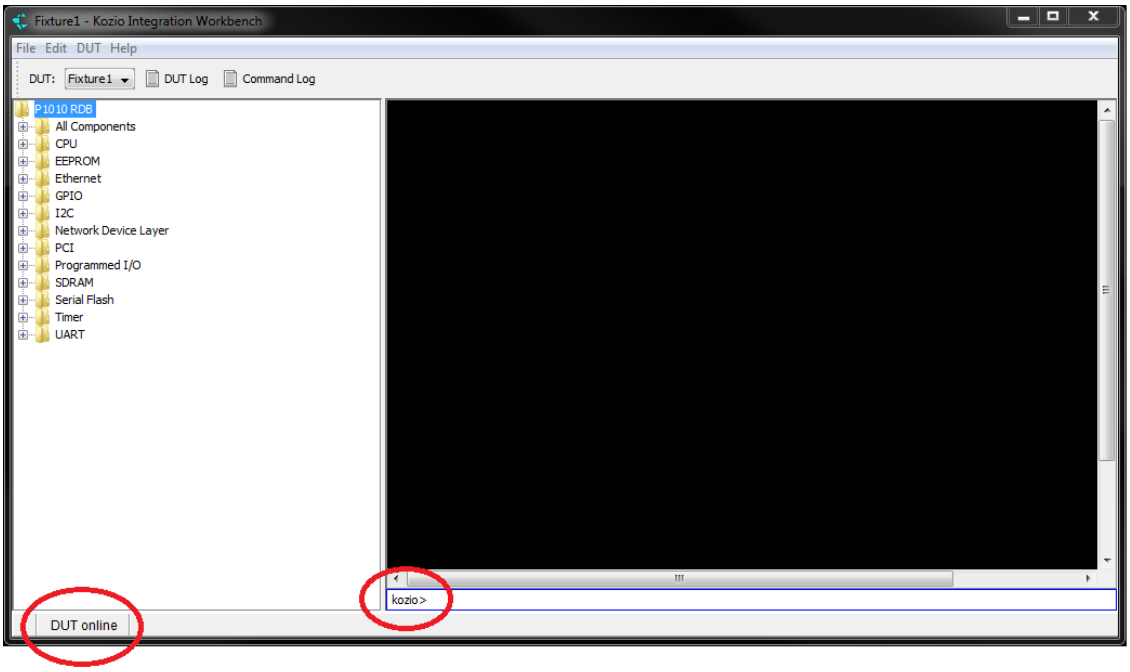


Figure 3.1: VTOS ready for command

To issue VTOS a command, type the name of the command at the Command Prompt, and press the ENTER key. The Command Prompt will be disabled while VTOS is processing your request. A new Command Line will become available when processing is complete and VTOS is ready for your next command.

You can practice issuing commands to VTOS by running various tests that exercise the memory controller and SDRAM on your evaluation platform. Try the following:

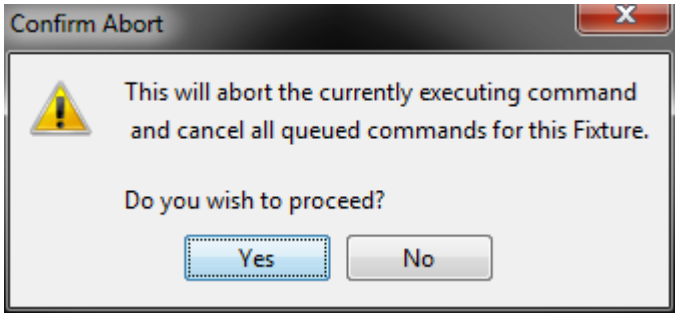
1. At the command prompt, type “**test.sdram.data0**”
   kozio> **test.sdram.data0**

2. Press the ENTER key to submit the command.

Notice that the command displays a progress bar in the lower right corner of the window to indicate that the test is running. When the test completes, VTOS will re-enable its command prompt to indicate that it is ready for your next command. You can experiment further with SDRAM tests using additional VTOS commands: “**test.sdram.data1**”, and “**test.sdram.address**”. As you use these commands, notice that the VTOS command interpreter does not distinguish between upper case and lower case letters. For instance, to the command interpreter, test.sdram.data0 and test.SDRAM.data0 are equivalent names for the command TEST.SDRAM.DATA0.

3.4 Interrupting commands

You can interrupt VTOS command execution at any time. Simply press the ESC key while the command is running and confirm. The command will stop running immediately, and VTOS will ready itself for your next command.



3.2: Confirm Abort Dialog

You can practice interrupting commands by following these steps:

1. At the command prompt, type "**begin test.sdram.data0 again**"

   kozio> **begin test.sdram.data0 again**

   This command will cause the SDRAM walking-zeroes test to loop infinitely. If you are curious, the words ***begin***, and ***again*** belong to the Kozio Scripting Language, **kScript**. This scripting language is introduced later in this tutorial, and is described completely in the ***Kozio kScript Reference***.

2. Press the ENTER key to submit the command.

3. Interrupt the command at any time by pressing ESC and confirming.

VTOS stops the SDRAM test and enables its command prompt, indicating it is ready for your next command.

Note: Interrupting memory tests may leave the data cache disabled. If your platform is running slower than expected, execute the command "CPU.DCACHE.ENABLE"

3.5 Using command history

When issuing commands to VTOS, it saves these commands in a *command history*. The VTOS command history facility gives you the ability to view, edit, and reissue the commands in this history. Using this facility, you can correct typing errors and significantly reduce repetitive keystrokes.

To invoke the command history facility, press the UP arrow key on your keyboard. This causes the latest command in the command history to be displayed at the command prompt. To reissue this command, simply press the ENTER key. To display the command prior to this one, press the UP arrow key again. You can continue backward through the command history using the UP

arrow key. Similarly, you can move forward through the command history using the DOWN arrow key. You can reissue any of these commands by simply pressing the ENTER key while the command is displayed at the command prompt.

To see this in action, try the following:

1. At the command prompt, type "**test.sdram.data0**" and press ENTER.

2. At the command prompt, type "**test.sdram.data1**" and press ENTER.

3. Press the UP arrow key. At the command prompt, VTOS will display the latest command in the command history: kozio> test.sdram.data1

4. Press the UP arrow key again. At the command prompt, VTOS will display the next latest command in the command history: kozio> test.sdram.data0

5. Press the ENTER key to reissue the command test.sdram.data0. VTOS runs this command again.

Continue to practice reissuing commands from the command history by using the UP and DOWN arrow keys to locate the commands, and then pressing the ENTER key to reissue these commands. As you practice, notice that only a finite number of commands are saved in the command history. Once the command history is full, the earliest command is discarded to make room for the latest command. Notice that once you reach the earliest command in the history, continuing to press the UP arrow key has no effect. Once you reach the latest command in the history, pressing the DOWN arrow key exits the command history facility and you are returned to an empty command prompt. Finally, notice that when you reissue a command, it becomes the latest command in the command history.

In addition to viewing and reissuing prior commands, you can also edit them, and then reissue them with these edits. Of course, you can also edit your current command before issuing it to VTOS. This ability to edit commands is handy for correcting typing mistakes and reducing repetitive keystrokes. To see it in action, try the following:

1. At the command prompt, type "**tezt.sdram.data0**". Notice that 'test' is misspelled.

2. Press ENTER to submit the command. VTOS will respond with a message indicating that it does not recognize the misspelled command.

   kozio> **tezt.sdram.data0**
   TEZT.SDRAM.DATA0 ? – No command with this name is defined!

3. Press the UP arrow key once. VTOS will display the command that has the typing mistake: kozio> tezt.sdram.data0

4. Use the LEFT and RIGHT arrow keys to move the cursor to the location of the typing error. Position the cursor to the right of the incorrect letter 'z'.

5. Press the BACKSPACE key to delete the letter ‘z’.

6. Type the letter ‘s’ to correct the misspelled word.

7. Press the ENTER key to submit the corrected command. VTOS will now run the SDRAM walking-zeroes test.

8. Press the UP arrow key once. VTOS will display the command that you have just run: kozio> test.sdram.data0

9. Use the BACKSPACE key to delete ‘0’ from the end of the word. Type **'1'** in its place: kozio> test.sdram.data**1**

10. Press the ENTER key to submit the command. VTOS will now run the SDRAM walking-ones test.

Continue using the SDRAM tests to practice using command history editing. As you practice, notice how the names of the SDRAM tests all share a common prefix: test.sdram. As you become familiar with command history editing, you will see that this naming convention allows you to run the various SDRAM tests with a minimal number of keystrokes. This common-prefix naming convention is used for all of the various VTOS test commands, and so it is worthwhile learning how to exploit it to reduce your keystrokes. You will see later in this lesson that you can also exploit this naming convention while using the online help system.

Table 3.1: Keyboard Shortcuts

Key	Function
ENTER	Issue command
UP	Prior command
DOWN	Next command
LEFT	Move cursor left
RIGHT	Move cursor right
BACKSPACE	Delete character left of cursor

3.6 Automating commands

Over time you will identify VTOS command sequences that you need to run frequently or share with others. Typing these command sequences directly at the command prompt can be tedious and error-prone. As an alternative, you can save these command sequences in text files on your host computer. You can then run these command sequences by submitting the file to VTOS via the “DUT/Execute Script” menu option or simply drag and drop the text file to the Integration Workbench Output Window or Command Line.

To see this technique in action, try the following:

1. Create a text file named lesson1.ksc on your host computer using your plain text editor of choice (Windows Notepad, UNIX vi, etc.). The extension .ksc indicates a Kozio script file; the .txt extension may also be used.

2. Enter the following contents into the text file:

   test.sdram.data0
   test.sdram.data1
   test.sdram.address

3. Save the text file to a known location on your host computer.

4. Use the Integration Workbench "DUT/Execute Script" menu option or drag and drop your new text file to the Integration Workbench Output Window or Command Line to execute the commands. VTOS will run the SDRAM walking zeroes test, then the SDRAM walking ones test, and finally the SDRAM address bus test.

Text files containing lists of commands are actually simple examples of kScript *scripts*. As your knowledge of VTOS commands and the kScript scripting language grows, your ability to create sophisticated automated command sequences will grow as well. You will explore the kScript scripting language in a later lesson.

3.7 Lesson review

1. **How does VTOS indicate it is ready to accept a new command?**
   VTOS enables its command prompt when it is ready to accept a new command.

2. **How can you issue commands to VTOS using your keyboard?**
   To issue a command to VTOS, type the name of the command at the command prompt and press ENTER on your keyboard.

3. **How can you interrupt a command that is running?**
   To interrupt a command, press the ESC key and confirm in the popup dialog box.

4. **What keys can you use to view items in the command history?**
   The UP arrow and DOWN arrow keys are used to view items in the command history.

5. **What keys can you use to edit and reissue commands in the command history?**
   The ENTER key is used to reissue commands from the command history. The LEFT, RIGHT and BACKSPACE keys are used to edit items in the command history.

6. **How can you automate frequently used command sequences?**
   To automate frequently used sequences of commands, save the sequences in a text file on the host computer. Use the Integration Workbench "DUT/Execute Script" menu option or simply drag and drop the text file to the Integration Workbench Output Window or Command Line to execute the commands.

4 Using Test Suites

In this lesson, you will explore VTOS test suites by using them to diagnose a simulated problem on the target. You will learn how to run test suites, how to examine the structure of test suites, how to interpret test suite results, and how to create your own custom test suites from existing VTOS tests.

4.1 About this lesson

In this lesson, you will learn how to do the following:

- Run groups of VTOS tests together as test suites.
- Determine how many tests in a test suite have passed, failed, or aborted.
- Exercise particular hardware components using predefined VTOS suites.
- Generate test suite reports.
- Examine the structure of test suites.
- Create custom test suites that run only the tests that you specify.

This lesson will take approximately 60 minutes to complete.

4.2 Running test suites

You can exercise hardware components on your evaluation platform using the extensive set of diagnostic tests included with VTOS. You can run the entire suite of tests with the single command "**test.example.long**" or by double clicking "Example Test Suite – long version" under "All Componnts/Suites" in the Command Tree. Issue the command test.example.long to VTOS now to see these tests in action.

kozio> **test.example.long**

While the tests are running, notice that each test displays its title, parameters, progress bar, and Passed/Failed/Aborted results. When the tests finish, VTOS displays final counts of the number of tests that have passed, failed, and aborted. You can use these final counts to verify that your evaluation platform has passed all of the tests.

The individual VTOS tests can be collected into groups of tests called *test suites*. Test suites themselves can also be grouped together to form master test suites. (test.example.long is an example of such a master test suite.) VTOS includes a number of predefined test suites that group tests together according to the hardware components they test. You can learn more about these predefined test suites by right-clicking on a Suite in the Command Tree and selecting "Help." Later in this lesson, you will learn how you can create custom test suites that group tests according to your specific needs.

The predefined VTOS test suites are named after the hardware components they test, following the pattern "test.*component*". You can run these test suites by directly issuing their names as

commands to VTOS at the Command Line or by double-clicking the associated Suite in the Command Tree. So, for example, you can issue the command "**test.sdram**" or double-click "SDRAM test suite (comprehensive)" in the Command Tree to run the test suite that tests your target's SDRAM components. Try using this test suite to exercise the SDRAM and memory controller on your evaluation platform. You can generate a results report for the last test run (of the current session) by issuing the command "**report test.sdram**" or right-clicking on "SDRAM test suite (comprehensive)" in the Command Tree and selecting "Report."

4.3 Dissecting test suites

You can examine the structure of test suites using the command "tree *suite-name*". (Replace the italicized text with the actual name of the test suite of interest.) Try examining the "**test.sdram**" test suite now by issuing the command "**tree test.sdram**" to VTOS at the Command Line or by right-clicking "SDRAM test suite (comprehensive)" in the Command Tree and selecting "Tree."

```
kozio> tree test.sdram
TEST.SDRAM (SDRAM Memory Test Suite - FULL)
    TEST.SDRAM.DATA0
    TEST.SDRAM.DATA1
    TEST.SDRAM.DATA.NOISE
    TEST.SDRAM.ADDRESS
    TEST.SDRAM.BYTE
    TEST.SDRAM.WORD
    TEST.SDRAM.BURST.NOISE
    TEST.SDRAM.BURST
    TEST.SDRAM.SSO
```

VTOS will display the name of the test suite, its title, and the names of the tests it contains. You can run each test separately by issuing the test's name as a command to VTOS at the Command Line. Try running some individual tests from the test.sdram test suite now. Notice that the test names share a common prefix: the name of the test suite to which they belong. This naming convention is followed by all VTOS tests and suites, and can save you many keystrokes when used together with the VTOS command history editing facility.

The structure of the test.sdram suite is simple because it does not contain other test suites. The test.example.short suite is more complex because it contains other test suites. Try examining the test.example.short test suite now by issuing the command "**tree test.example.short**" to VTOS at the Command Line or by right-clicking "Example Test Suite - short version" in the Command Tree and selecting "Tree."

kozio> **tree test.example.short**

The actual structure of the “test.example.short” suite will vary across evaluation platforms.

```
TEST.EXAMPLE.SHORT (VTOS Test Suite Example: short version)
    TEST.UART0 (UART 0 Test Suite)
        TEST.UART0.DATA0
        TEST.UART0.DATA1
        TEST.UART0.INT
    TEST.SDRAM.QUICK (SDRAM Memory Test Suite - Quick)
        TEST.SDRAM.DATA0
        TEST.SDRAM.DATA1
        TEST.SDRAM.DATA.NOISE
        TEST.SDRAM.ADDRESS
        TEST.SDRAM.SSO
    TEST.FLASH.QUICK (Flash Test Suite - quick test of all devices)
        TEST.FLASH0.QUICK (Flash Test Suite - Quick test of Flash
Device 0)
            TEST.FLASH0.WALK0
            TEST.FLASH0.ERASE
    TEST.SLIC.SPI (SLIC Register Access Test Suite)
        TEST.SLIC0.SPI
        TEST.SLIC1.SPI
    TEST.PCI (PCI Test Suite)
        TEST.PCI.CONFIG
    TEST.ENET.PHY.SCAN
```

Notice that VTOS displays the contents of the test suite using several levels of indentation. These indentation levels depict an outline of the test suite’s structure. At the first level in the outline, VTOS lists the items contained in the test suite. Since these items happen to be test suites, they also contain items, which are displayed at the next level in the outline – and so on. Altogether, the display resembles a formal outline of the test suite, with test suite names serving as category headings, and test names serving as subtopics under these categories.

Practicing examining test suites using these steps to examine the “test.example.short” suite:

1. Issue the command “**tree test.example.short**” to VTOS. VTOS will display an outline of the “test.example.short” suite.

2. Choose a test suite name from those listed at the first level in the outline.

3. Issue the name of your chosen test suite as a command to VTOS. VTOS will run the tests contained in your chosen test suite.

4. Issue the command “tree *suite-name”*, replacing the italicized text with the name of your chosen test suite. VTOS will display an outline of your chosen test suite.

5. Choose a test from those contained in the suite you have chosen. These tests will be listed below your chosen suite in the outline.

6. Run your chosen test by issuing its name as a command to VTOS. VTOS will run your chosen test.

4.4 Creating custom test suites

You can create your own custom test suites “on the fly” to suit your specific needs. To create a custom suite, you only need to supply VTOS with a title for the suite, a command name for the suite, and a list of tests or suites for the custom suite to run. Your custom suite will automatically inherit all of the display and report features of the predefined VTOS suites.

```
\ Comment:  Create a group of tests or suites called ‘custom-test-group-name’.
\ Comment:  The group contains the n items listed above it.

=> test-name or suite-name
=> test-name or suite-name
=> test-name or suite-name
n TESTS custom-test-group-name

\ Comment:  Create a test suite called ‘custom-suite-name’.
\ Comment:  The suite contains the items in ‘custom-test-group-name’.

S" Suite Title" => custom-test-group-name SUITE custom-suite-name
```

Figure 4.1: Custom Suite Template

Use this template as a guide for creating your own custom test suites. Replace the italicized text with your custom information. The => symbol (EQUALS-GREATER-THAN) is part of the template, and must be included in your custom file. The space characters after => and s" also must be included in your custom file. Your test group name and suite name cannot include space characters.

You can create custom test suites using a plain text editor on your host computer. To create a custom suite, create a file on your host computer, using the custom suite template in “Figure 4.1: Custom Suite Template” as a guide. Substitute the italicized text in the template with your custom information. Finally, send this text file to VTOS via the “DUT/Execute Script” menu option or simply drag and drop the text file to the Integration Workbench Output Window or Command Line. You can run your custom suite immediately after submitting it to VTOS by issuing your chosen test suite name as a command to VTOS.

You can practice creating custom scripts using the following steps to create a custom test suite called test.custom. This custom suite will run the test “test.flash.erase”, followed by all tests in the suite “test.sdram”, and then finish with the test “test.flash.walk0”.

1. Using your text editor of choice (Notepad, vi, etc), create a file on your host computer called custom.ksc. The extension .ksc indicates a Kozio script file; the .txt extension may also be used.

2. Enter the following content into the text file, using the custom script template as a guide. Carefully recreate the given text, including all space characters. (Note the space characters after the symbols S", ", and =>)

```
=> test.flash.erase
=> test.sdram
=> test.flash.walk0
3 TESTS group.custom

S" My custom suite" => group.custom SUITE test.custom
```

3. Save the text file to a known location on your host computer.

4. The test script can be easily run by sending the text file to VTOS via the "DUT/Execute Script" menu option or simply drag and drop the text file to the Integration Workbench Output Window or Command Line. Once submitted to VTOS, the new suite is immediately available. You can confirm this by entering "**tree test.custom**" at the Command Line.

```
kozio> tree test.custom
TEST.CUSTOM (My custom suite)
    TEST.FLASH0.ERASE
    TEST.SDRAM (SDRAM Memory Test Suite - FULL)
        TEST.SDRAM.DATA0
        TEST.SDRAM.DATA1
        TEST.SDRAM.DATA.NOISE
        TEST.SDRAM.ADDRESS
        TEST.SDRAM.BYTE
        TEST.SDRAM.WORD
        TEST.SDRAM.BURST.NOISE
        TEST.SDRAM.BURST
        TEST.SDRAM.SSO
    TEST.FLASH0.WALK0
```

5. Run the custom suite by issuing its name, "**test.custom**", as a command to VTOS.

You may be curious about the special symbols contained in the custom suite template. These symbols are elements of the Kozio scripting language, kScript. In this language, the S" (S-QUOTE) and " (QUOTE) symbols are used to create *strings*, character sequences that are stored together as a unit. The symbol S" (S-QUOTE) marks the start of the character sequence, and the symbol " (QUOTE) marks the end of the sequence. The characters between S" and " are stored together to form the string. The symbol => (EQUALS-GREATER-THAN) is used to create a *reference* to an item. This reference can then be used to *refer* to the item without actually *invoking* the item. In the template, => is used in front of the test names to *refer* to the tests without actually running them.

4.5 Lesson review

1. **How can you determine the number of tests in a suite that have passed, failed, or aborted?**

   You can determine the number of tests in a suite that have passed, failed or aborted using the final counts VTOS displays when the suite has finished running. The final counts show the number of tests that have passed, failed, and aborted.

2. **How can you exercise particular hardware components using predefined VTOS test suites?**

   You can exercise a particular hardware component using the predefined VTOS test suite that is named after the component. The VTOS test suites follow the naming convention "test.*component*". For instance, you can exercise flash memory components using the suite "test.flash".

3. **How can you examine the structure of a given test suite?**

   You can examine the structure of a given test suite using the command "tree *suite-name*". VTOS will display an outline that depicts the test suite's structure.

4. **How can you generate a report of the latest results for a given test suite?**

   You can generate a report of the latest results for a given test suite by issuing the command "report *suite-name*" to VTOS at the Command Line or by right-clicking on the Test Suite in the Command Tree and selecting "Report." VTOS will display an outline of the test suite, with the latest results of its tests displayed next to their name.

5. **How can you create custom test suites that run only the tests you specify?**

   You can create custom test suites using a plain text editor on your host computer. Create a file, using the custom suite template in "Figure 4.1: Custom Suite Template" as a guide. Supply your custom information in the template, and send the file to VTOS via the "DUT/Execute Script" menu option or simply drag and drop the text file to the Integration Workbench Output Window or Command Line.

5 Creating Custom Scripts

In this lesson, you will learn about the VTOS scripting language, kScript, by using it to create a custom test that exercises your evaluation platform's DMA hardware. You will familiarize yourself with the basic principles of the VTOS command interpreter, and learn many essential constructs available through the scripting language.

5.1 About this lesson

In this lesson, you will learn how to do the following:

- Use the interpreter's data stack.
- Define new commands for the interpreter.
- Read and write memory locations.
- Create named constants.
- Reserve a memory location to store variable data.
- Control the execution sequence of commands.
- Use local variables to simplify your new command definitions.
- Create custom diagnostic tests.

This lesson will take approximately 60 minutes to complete.

5.2 Seeing it all work

VTOS understands a programming language called kScript. You can use this scripting language to program VTOS with custom tests that suit your needs. An example script written in kScript is provided at the end of this lesson.

The example script creates a custom test that exercises your target's DMA hardware using a varying data pattern. If your evaluation platform supports DMA, you can upload this script to VTOS and run the custom test to see it in action. Simply drag and drop the script into Integration Workbench and run the script on the command line.

The example script will be used throughout this lesson to illustrate key features of kScript.

5.3 Understanding scripts

You can submit text to VTOS either by entering it manually at the command prompt, or by dragging and dropping the script into Integration Workbench. When you submit text to VTOS, VTOS *interprets* your text. VTOS knows how to interpret two kinds of items: *command names* and *numbers*.

VTOS uses white space characters (SPACE, TAB, or NEW-LINE) to mark the boundaries of (delimit) command names and numbers in your submitted text. (You use a similar rule to delimit

words when you interpret written English text.) VTOS requires only a single white space character between language items in your scripts. However, you are encouraged to use white space liberally in your scripts to visually organize them for human readers. Study the example script now (see excerpt in Figure 5.1), and notice how it uses white space to visually separate and organize its components.

```
  \ verify each word in memory range contains pattern
  \
  \ Return TRUE on the stack if the entire memory range was verified.
  \ Return FALSE on the stack if a word didn't match the pattern
: pattern.verify { pattern address length | result }
   length bytes.to.words -> length
   TRUE -> result

   length 0
   do
       address read
       pattern <> if
           \ value does not match pattern
           FALSE -> result
           leave  \ exit loop
       then
       address word+ -> address
   loop

   result \ place result on stack
;
```

Figure 5.1: Excerpt from custom DMA test showing the visual use of white space

Command names are sequences of letters, digits, punctuation characters, or symbols. *Command names*, not surprisingly, are used to name *commands*. *Commands* are procedures, and *command names* are names for these procedures. VTOS interprets command names in your scripts by running the commands that they name. Some examples of potential command names are underlined in the following text:

\ pattern.verify pattern | : ; { } -> do loop <> word+

Numbers are sequences of digits. VTOS interprets numbers in your scripts by converting them into values and placing these values onto the *data stack* (more about this in a moment). VTOS has a few special rules for interpreting numbers. Numbers that are prefixed with the – (MINUS) character are converted into negative values. Numbers that are prefixed with the characters 0x (ZERO-x) are interpreted as hexadecimal numbers. Some examples of numbers are given in the following text (hexadecimal numbers are underlined).

12 –5 0x100 0xc2000000 0xABCD –0x1E

VTOS also supports a special format for interpreting IP address values, used with Ethernet testing. Numbers that are prefixed with the characters IP/ (IP-SLASH) denote an IP address value. Some examples of IP address values are given in the following text.

IP/192.168.23.2 ip/255.255.255.0 ip/12.64.101.210

VTOS scripts are simply sequences of command names and numbers. When you submit the name of a command to VTOS at the command prompt, you are actually submitting a very simple script for VTOS to interpret: a script consisting of a single command name. VTOS interprets this simple script by running the command that has that name.

5.4 Using comments

You can annotate your scripts using *comments*. You are encouraged to use comments judiciously in your scripts to make them easily understood by others (and yourself!).

You can add comments to your scripts using the command \ (BACKSLASH) or the command // (SLASH-SLASH). The \ and // commands cause any remaining characters in the text line to be ignored by VTOS. To add comments to your script, simply follow the \ command with the text that comprises your comment. VTOS will not attempt to interpret your comment's text, and will resume interpreting your script at the beginning of the next text line.

Note that \ and // are a command names, and must be delimited by white space. Observe the following different text interpretations by VTOS:

kozio> **\ My comment**
kozio> **\My comment**
\MY ? - No command with this name is defined!

Study the example script now (see excerpt in Figure 5.2), and notice how it uses comments to describe important features.

```
\ SECTION 1: constants and variables ***************************

          0 constant  const.dma.channel  \ DMA channel
 0xc2000000 constant  const.dma.source   \ DMA source address
 0xc4000000 constant  const.dma.dest     \ DMA destination address
        256 value     $dma.length        \ DMA transfer length (bytes)


\ SECTION 2: pattern fill/verify commands **********************

  \ convert word count on stack to a byte count
 : words.to.bytes 4 * ;
```

Figure 5.2: Excerpt from custom DMA script showing use of comments

5.5 Using the data stack

VTOS manages an area in memory called *the data stack*. You can use the data stack as a scratch area for storing intermediate computations.

VTOS manages items on the data stack using a Last-In-First-Out (LIFO) policy. The data stack can be compared to a stack of bricks. You build a stack of bricks by placing bricks on top of one another. When you add the first brick to the stack, it is the top brick in the stack. When you add another brick to the stack, this brick becomes the top brick in the stack, and the original brick becomes the second brick from the top. As you continue adding bricks to the stack, the last brick you add becomes the top brick; the brick that was formerly on top becomes the second brick; the brick that was formerly the second brick becomes the third – and so on. Now, when you remove bricks from the stack, you begin by removing the top brick from the stack. The brick that was second from the top becomes the top brick; the brick that was third from the top becomes the second brick – and so on. Notice that as you remove bricks from the stack, the last brick placed on the stack (Last-In) is the first brick removed (First-Out).

You can remove all items from the data stack using the command 0sp (ZERO-sp). Run this command now to begin with an empty data stack.

kozio> **0sp**

Recall that VTOS interprets numbers in your scripts by converting these numbers into values, and placing the values onto the data stack. So, you can place values onto the stack by simply giving VTOS numbers to interpret. You can see this in action by trying the following now:

```
kozio> 1492
kozio> .s
Stack<10> 1492

kozio> 1776
kozio> .s
Stack<10> 1492 1776

kozio> .
1776

kozio> .s
Stack<10> 1492
```

The .s (PERIOD-S) command displays the stack. Notice that the .s (PERIOD-S) command displays the bottom of the stack first (left-most), and the top of the stack last (right-most). Notice that the. (PERIOD) command removes the top value from the stack and displays it.

Generally, you put values on the stack to supply inputs to commands that *operate* on the values. And generally, commands put the results of their operations on the stack for use by other commands. You can use the + (PLUS) command, for instance, to calculate the sum of two values you have put on the stack. The command removes your values from the stack, calculates their sum, and puts the sum on the stack. Similarly, you can calculate the product of two values using the * (ASTERISK) command. Follow the steps below to use these commands to calculate the expression: 704 (1776 + 1492)

1. Remove all items from the data stack.

   ```
   kozio> 0sp
   ```

2. Calculate the sum in the expression.

   ```
   kozio> 1776 1492 +
   ```

   VTOS interprets the text from left to right. First, VTOS interprets the text 1776, converting it to its numeric value and placing this value on the stack. Next, VTOS interprets the number 1492, converting it to a value and placing this value on the stack. Finally, VTOS interprets the command name + by running its associated command.

3. Display the stack to see this intermediate result.

   ```
   kozio> .s
   Stack<10> 3268
   ```

   Notice that the + command has removed its inputs (1776 and 1492) from the stack, and has placed their sum on the stack.

4. Calculate the product in the expression.

   kozio> **704 ***
   kozio> **.**
   2300672

   First, VTOS interprets the number 704, converting it to a value and placing this value on the stack. (The stack now has two items. The bottom of the stack holds the result of the calculation performed in step 3.) Next, VTOS interprets the command name * by running its associated command. The * command removes its two inputs from the stack, and places their product on the stack.

- You may recognize the notation in step 2 above as *postfix* notation, a style of notation used by popular calculators. Postfix is a notation in which operators are placed to the right of their operands, and operations of higher precedence are listed to the left of operations of lower precedence. Postfix notation allows expressions to be notated unambiguously without parentheses. Try translating expression 5 (4 + 3 (1 + 2)) into postfix notation now. Then, submit your translation to VTOS to verify it.

  kozio> **1 2 + 3 * 4 + 5 * .**
  65

5.6 Defining new commands

You can define new commands that perform the procedures you specify, and run these commands using the names you have given them.

You can define new commands using the : (COLON) and ; (SEMI-COLON) commands. You use these commands to mark the beginning and end of a command's definition.

```
: new-command-name
   new command's procedure ...
;
```

The : and ; commands create a command with the name *new-command-name*. You can use any sequence of command names and numbers to define your new command's procedure. Once you have defined your new command, you can use its name *new-command-name* to invoke its procedure. VTOS will interpret *new-command-name* by running the procedure you have defined. You can see this in action by performing the following steps to create new commands called my.sdram1 and my.sdram2 and my.tests.

```
kozio> : my.sdram1 test.sdram.data1 test.sdram.data0 ;
kozio> my.sdram1
kozio> : my.sdram2 test.sdram.byte ;
kozio> my.sdram2
kozio> : my.tests  my.sdram1 my.sdram2 ;
kozio> my.tests
```

Notice how the my.tests command uses the my.sdram1 and my.sdram2 commands in its definition. You can use your commands to compose new commands, building layers of commands that perform evermore-sophisticated procedures.

Your commands can take inputs from the data stack, perform computations using these inputs, and leave the results of these computations on the stack. You can see this in action by following these steps to create a command called add-four that adds four to the value on the top of the data stack:

```
kozio> 0sp
kozio> : add-four 4 + ;
kozio> 1
kozio> .s
Stack<10> 1
kozio> add-four .s
Stack<10> 5
kozio> add-four add-four .s
Stack<10> 13
```

You should now be able to understand the words.to.bytes and bytes.to.words commands defined by the example script (see excerpt in Figure 5.3). Study these commands to strengthen your understanding of commands and the data stack.

```
 \ convert word count on stack to a byte count
: words.to.bytes 4 * ;

 \ convert byte count on stack to a word count, rounding up
: bytes.to.words 3 + 4 /  ;
```

Figure 5.3: Excerpt from custom DMA test showing the definition of some commands that use the data stack

5.7 Using input and output functions

You can read and write addresses in your processor's virtual address space. You can perform these operations using different access widths.

You can read from an address using the read (read.halfword,read.byte) command. The read command reads the word stored at the given address, and places this value on the data stack. The read command requires an input from the stack – the address of the location to read.

You can write a value to a location using the write (write.halfword, write.byte) command. The write command requires two inputs from the stack – the address to write and the value to write. Note that the write command assumes that the address is the top item on the stack, and the value is the next item on the stack.

You can practice using the read and write commands now by following these steps:

1. Display 8 words starting at address 0xC2000000.

```
kozio> 0xc2000000 32 dump
0xC2000000 : 00000000 00000000 00000000 00000000
0xC2000010 : 00000000 00000000 00000000 00000000
```

2. Write a pattern to addresses 0xC2000000 and 0xC4000000.

```
kozio> 0x55555555 0xc2000000 write
kozio> 0xaaaaaaaa 0xc2000004 write
kozio> 0xc2000000 32 dump
0xC2000000 : 55555555 AAAAAAAA 00000000 00000000
0xC2000010 : 00000000 00000000 00000000 00000000
```

3. Read the value at address 0xC2000000, and place the value on the stack.

```
kozio> 0xc2000000 read
```

4. Display contents of stack in hexadecimal notation.

```
kozio>.hex
0x55555555
```

You can see real examples of read and write being used in the definitions of the pattern.fill and pattern.verify routines of the example script (see excerpt in Figure 5.5). These commands use the read and write commands in repetitive loops to fill memory ranges with patterns and verify that memory ranges contain patterns.

5.8 Naming constants

You can give descriptive names to important constants. You can then use these descriptive names instead of using numbers to specify these constants. This can make your scripts easier to understand, and easier to maintain.

You can give a name to a constant using the command constant *constant-name*. Your *constant-name* parameter can be any sequence of characters, digits, punctuation, and symbols. The constant command creates a command with the name *constant-name* that can be used in place of a given number in your script.

The constant command requires an input from the data stack – the value that you want to associate with *constant-name*. Typically, you supply this value to the constant command directly by preceding it with a number in your script.

Once you have defined a constant, you can access its value by simply typing the constant's name. Observe in the following example that you can use constants just as you numbers in your scripts: (If you have not uploaded the custom DMA script to VTOS, do so now. The script defines constants that are used by this procedure.)

1. Place the value of the constant const.dma.source on the stack and display it.

```
kozio> const.dma.source
kozio> .hex
0xC2000000
```

2. Use the DMA source constant as part of a dump command.

```
kozio> const.dma.source 32 dump
0xC2000000 : 55555555 AAAAAAAA 00000000 00000000
0xC2000010 : 00000000 00000000 00000000 00000000

kozio> 0x11111111 const.dma.source write
kozio> const.dma.source 32 dump
0xC2000000 : 11111111 AAAAAAAA 00000000 00000000
0xC2000010 : 00000000 00000000 00000000 00000000
```

You can see real examples of the constant command in the example script (see excerpt in Figure 5.4). Notice that the script assigns names to DMA source and destination addresses used by the test: const.dma.source and const.dma.dest. These addresses are then referred to by these names in the remainder of the script, making the script easier to understand. Notice that you can easily change the addresses used by the test by simply changing the numbers that assign values to these names.

```
\ SECTION 1: constants and variables ***************************

          0 constant  const.dma.channel  \ DMA channel
 0xc2000000 constant  const.dma.source   \ DMA source address
 0xc4000000 constant  const.dma.dest     \ DMA destination address
        256 value     $dma.length        \ DMA transfer length (bytes)
```

Figure 5.4: Excerpt from custom DMA test showing use of constants and variables

5.9 Creating variables

You can reserve and name memory locations for storing variable data. You can then refer to these locations by name.

You can reserve and name a memory location using the command "**value *variable-name***". Your *variable-name* parameter can be any sequence of characters, digits, punctuation, and symbols. The value command creates a command with the name *variable-name* that can be used to refer to the memory location you have reserved.

The value command requires an input from the data stack – the initial value that you want to store to the memory location you are reserving. Typically, you supply this initial value to the value command directly by preceding it with a number in your script.

You access the value of a variable by typing the variable's name directly at the command prompt. To change the value of a variable, you use the command "**->**" (DASH-GREATER-THAN). Practice creating a variable and changing its value now:

1. Create a new variable called *$dma.length* with an initial value of 256.

   ```
   kozio> 256 value $dma.length
   ```

2. Place the current value of the variable *$dma.length* on the stack and display it to the console.

   ```
   kozio> $dma.length
   kozio> .hex
   0x100
   ```

3. Modify the value of *$dma.length* and confirm the change.

   ```
   kozio> 0x80000 -> $dma.length
   kozio> $dma.length
   kozio> .hex
   0x80000
   ```

You can see a real example of the value command in the example script (see excerpt in Figure 5.4). Notice that the script assigns the name *$dma.length* to a memory location that holds the DMA transfer length used by the test. The script makes this value a variable so that the transfer length used by the test can be changed by changing the value stored in this location. Notice that this allows the test behavior to be changed without changing the script. On the other hand, changing the source and destination addresses used by the test requires changing the script, because they have been defined as constants.

5.10 Managing control flow

You can control the execution sequence of commands in your commands' procedures. Your commands can use *loop*s to accomplish repetitive procedures. Your commands can perform *branches* to execute procedures based on conditions.

You can use the "**begin**" and "**again**" commands together to mark the beginning and end of command sequences that you want to repeat endlessly.

```
begin
      repeated command sequence
again
```

You can see these loop commands in action by issuing the following command to VTOS to repeat the command test.sdram.data0 endlessly. (You can still interrupt the loop using ESC).

```
kozio> begin test.sdram.data0 again
```

You can use the do and loop commands together to mark the beginning and end of command sequences that you want to repeat a limited number of times.

```
command sequence
do
      repeated command sequence
loop
command sequence
```

The do command requires two values from the data stack – a loop limit and a loop index. These values are placed on the stack by the command sequence preceding the do command. (The do command assumes that the loop index is the top value on the stack.)

If the loop index is less than the loop limit, then the *repeated command sequence* is executed, and the loop index is incremented. This process continues until the loop index equals the loop limit. Then, execution proceeds to the command sequence following the loop command.

You can make use of the loop index in the *repeated command sequence* to compute values. You access the loop index using the "**I**" command. The "**I**" command places the current value of the loop index onto the top of the stack. You can see the do, loop, and I command in action by issuing the following commands to VTOS to compute the first five multiples of the number ten:

```
kozio> 5 0 do I 1 + 10 * loop
kozio> .s
Stack<10> 10 20 30 40 50
```

You can see real examples of the do and loop commands being used in the example script (see excerpt in Figure 5.5).

You can create branches using the "**if**" and "**then**" commands. You use these commands together to mark the beginning and end of a conditional command sequence that executes when a condition has been met.

```
command sequence
if
      conditional command sequence
then
command sequence
```

Optionally, you can use the "**else**" command together with the "**if**" and "**then**" commands to specify an alternate command sequence that executes when the condition has not been met.

```
        command sequence
        if
              conditional command sequence
        else
              alternate command sequence
        then
        command sequence
```

The "**if**" command requires an input from the data stack – a value indicating whether a condition has been met. This value is placed onto the stack by the command sequence preceding the "**if**" command. If any bit in this value is non-zero, the *conditional command sequence* is executed. Then, execution proceeds to the command sequence following the "**then**" command. If all bits in this value are zero, the *alternate command sequence* is executed. Then, execution proceeds to the command sequence following the "**then**" command.

```
  \ write each word in memory range with pattern
: pattern.fill  { pattern address length }
   length bytes.to.words -> length

   length 0
   do
       pattern address write
       address word+ -> address
   loop
;

  \ verify each word in memory range contains pattern
  \
  \ Return TRUE on the stack if the entire memory range was verified.
  \ Return FALSE on the stack if a word didn't match the pattern
: pattern.verify { pattern address length | result }
   length bytes.to.words -> length
   TRUE -> result

   length 0
   do
       address read
       pattern <> if
           \ value does not match pattern
           FALSE -> result
           leave  \ exit loop
       then
       address word+ -> address
   loop

   result \ place result on stack
;
```

Figure 5.5: Excerpt from custom DMA test showing use of control flow commands

You can use the value comparison commands "=" ("is equal to") or "<>" ("is not equal to") together with the "**if**" and "**then**" commands to execute command sequences when two values are equal or not equal. You can see a real example of this usage in the example script (see excerpt in Figure 5.5). Notice that the pattern.verify command uses the "<>"**,** "**if**"**,** and "**then**" commands to detect pattern mismatches.

5.11 Using local variables

You can use *local variables* in your commands to simplify managing the intermediate values in your commands' computations. You use the "**{**" (LEFT-CURLY-BRACE), "|" (PIPE), and "**}**" (RIGHT-CURLY-BRACE) commands to define local variables for your command.

{ *stack-initialized-variables ...* | *zero-initialized-variables ...* }

Stack initialized local variables are assigned values from the data stack – input parameters sent to your command. *Zero initialized* local variables are a temporary memory location reserved to hold a value needed by your command and are automatically initialized to the value zero. You specify multiple local variables by separating the variable names with spaces. You can define stack-initialized local variables only, zero-initialized local variables only, or both. If you only need stack-initialized local variables, the use of the "|" command is optional. However, if you define only zero-initialized local variables, you must precede the first variable name with the "|" command.

```
: new-command-1
{ param-1 param-2 | zero-init-1 zero-init-2 }
     command procedure
;

: new-command-2
{ param-1 param-2 param-3 }
     command procedure
;

: new-command-3
{ | zero-init-1 }
     command procedure
;
```

You access the value of a local variable by typing the local variable's name directly in your script. To change the value of a local variable, you use the command "**->**" (DASH-GREATER-THAN). Try the following steps below to see how to use local variables:

1. Define a new command that has one stack-initialized local variable and one zero-initialized variable. Type the following at the VTOS prompt.

```
kozio> : sum { n | accum }
1 n + -> n
n 0 do
    accum i +
    -> accum
loop
accum
;
```

2. Execute the command and display the results.

```
kozio> 3 sum
kozio> .
6
kozio> 25 sum
kozio> .
325
```

If your command uses local variables, the "**{**" command must be the first command in your new command's procedure. You can see further examples of both initialized and uninitialized local variables in the definitions of the "pattern.fill" and "pattern.verify" routines of the example script (see excerpt in Figure 5.5). Notice that using local variables enhances the readability of your scripts.

5.12 Creating custom tests

You can create custom diagnostic tests using commands that manage your tests' control flow. Your custom tests will automatically inherit the display characteristics of predefined VTOS tests, including progress bars and Pass/Fail/Abort indicators.

You use the "**test.begin**", "**test.while**", and "**test.repeat**" commands to define your test procedure.

```
s" Test-Name-Text"
test-iterations test.begin
    command-sequence-1
    test.while

    command-sequence-2
    test.while

     ...

    command-sequence-N
    test.while
test.repeat
```

The *test procedure* is defined as the entire sequence of commands between the "**test.begin**" and "**test.repeat**" commands. The "**test.begin**" command requires three inputs from the data stack – a string specifying the name of the test (which occupies two inputs on the data stack) and a value indicating how many times to run the *test procedure*.

The "**test.while**" command conditionally exits the test procedure and requires one input from the data stack – a value indicating whether the preceding *command-sequence* was successful. If any bit in this value is non-zero, execution continues to the next *command-sequence* (or to the "**test.repeat**" command). If all bits in this value are zero, the test is failed and execution continues following the "**test.repeat**" command.

When execution reaches the "**test.repeat**" command (because all intermediate "**test.while**" checks within the *test procedure* passed), if the test loop index is less than the test iterations, then the *test procedure* is executed again. This process continues until the test loop index equals the test iteration count. Then, VTOS displays a PASSED indication and execution proceeds to the command sequence following the "**test.repeat**" command.

You can access the current test loop index in the *test procedure* using the test.loop.count command. The "**test.loop.count**" command places the current value of the test loop index onto the top of the data stack.

You can see a real example of a custom diagnostic test in the example script (see excerpt in Figure 5.6). As you study this example, notice that the *test procedure* will run 32 times. Each pass through the *test procedure* performs a DMA memory transfer with a new data pattern, verifying the source and destination memory areas before and after the DMA transfer. You can see that by relying on previously defined commands, the actual *test procedure* becomes quite simple.

```
\ SECTION 4: Custom DMA Pattern Test ***************************
: test.dma.pattern { | pattern }

   s" VTOS Tutorial Lesson 3: Custom DMA test"
   32 TEST.BEGIN

    1 TEST.LOOP.COUNT lshift -> pattern  \ shift bit left

                pattern        fill.and.verify.source
    TEST.WHILE pattern invert fill.and.verify.destination
    TEST.WHILE                ...dma.transfer...
    TEST.WHILE pattern        verify.destination
    TEST.WHILE pattern        verify.source
    TEST.WHILE

   TEST.REPEAT
 ;
```

Figure 5.6: Excerpt from custom DMA test showing use of test procedure commands

Now, run the custom DMA test using the command test.dma.pattern. Observe the differences in the test execution time as you experiment with changing the value of the variable *$dma.length* and rerunning the test.

5.13 Custom DMA script

```
\    This script creates a test named 'test.dma.pattern' that
\    tests DMA hardware using a varying data pattern.
\
\    The test performs these steps:
\
\     (1) Fill DMA source with pattern
\     (2) Verify DMA source contains pattern
\     (3) Fill DMA destination with inverse of pattern
\     (4) Verify DMA destination contains inverse of pattern
\     (5) Initiate DMA transfer. Wait for transfer to complete
\     (6) Verify DMA destination contains pattern
\     (7) Verify DMA source contains pattern
\
\  The test stops and reports a failure if an error is detected
\  after any of these steps.
\
\  The test iterates 32 times, varying the data pattern on each
\  iteration using a "walking 1" data pattern.
\
\  The DMA source and destination addresses are fixed.
\  The DMA data length is variable, with a default of 256 bytes.
\  You can change the data length like this:
\
\   Example: use 8 KB transfer length
\
\      kozio> 8192 -> $dma.length
\      kozio> test.dma.pattern

\ SECTION 1: constants and variables ***************************

 0xc2000000 constant  const.dma.source   \ DMA source address
 0xc4000000 constant  const.dma.dest     \ DMA destination address
        256 value     $dma.length        \ DMA transfer length (bytes)


\ SECTION 2: pattern fill/verify commands **********************

  \ convert word count on stack to a byte count
 : words.to.bytes 4 * ;

  \ convert byte count on stack to a word count, rounding up
 : bytes.to.words 3 + 4 /  ;

  \ write each word in memory range with pattern
 : pattern.fill  { pattern address length }
    length bytes.to.words -> length

    length 0
    do
        pattern address write
        address word+ -> address
    loop
```

```
;

 \ verify each word in memory range contains pattern
 \
 \ Return TRUE on the stack if the entire memory range was verified.
 \ Return FALSE on the stack if a word didn't match the pattern
: pattern.verify { pattern address length | result }
   length bytes.to.words -> length
   TRUE -> result

   length 0
   do
       address read
       pattern <> if
           \ value does not match pattern
           FALSE -> result
           leave  \ exit loop
       then
       address word+ -> address
   loop

   result \ place result on stack
;

\ SECTION 3: DMA source/destination fill/verify commands *******

 \ verify DMA destination address range contains pattern
: verify.destination  { pattern }
       pattern const.dma.dest $dma.length  pattern.verify
;

 \ fill DMA destination address range with pattern, and verify
: fill.and.verify.destination  { pattern }
       pattern const.dma.dest $dma.length  pattern.fill
       pattern verify.destination
;

 \ verify DMA source address range contains pattern
: verify.source { pattern }
       pattern const.dma.source $dma.length  pattern.verify
;

 \ fill DMA source address range with pattern, and verify
: fill.and.verify.source  { pattern }
       pattern const.dma.source $dma.length  pattern.fill
       pattern verify.source
;

 \ start DMA transfer from DMA source to DMA destination,
 \ using VTOS DMA transfer routine to manage the transfer
 \
 \ Return TRUE on the stack indicating if the DMA transfer
 \ completed successfully. Return FALSE on the stack if
 \ the DMA transfer timed out.

: ...dma.transfer... { | length }
```

```
       $dma.length bytes.to.words words.to.bytes -> length
       const.dma.channel const.dma.source const.dma.dest length dma.transfer
       ERRORLEVEL CONST.TEST.PASSED  =
 ;

\ SECTION 4: Custom DMA Pattern Test ***************************
 : test.dma.pattern { | pattern }

   s" VTOS Tutorial Lesson 3: Custom DMA test"
   32 TEST.BEGIN

    1 TEST.LOOP.COUNT lshift -> pattern  \ shift bit left

                pattern        fill.and.verify.source
    TEST.WHILE pattern invert fill.and.verify.destination
    TEST.WHILE                ...dma.transfer...
    TEST.WHILE pattern        verify.destination
    TEST.WHILE pattern        verify.source
    TEST.WHILE

   TEST.REPEAT
 ;
```

5.14 Lesson review

1. **What characters delimit command names and numbers in your scripts?**

   White space characters (SPACE, TAB, or NEW-LINE) delimit command names and numbers in your submitted text and scripts.

2. **How can you annotate your scripts, making them easier to understand?**

   You can add comments to your scripts using the command \ (BACKSLASH) followed by the text that comprises your comment.

3. **How can you supply inputs to commands?**

   You can put values on the data stack to supply inputs to commands that operate on the values. VTOS manages items on the data stack using a Last-In-First-Out (LIFO) policy.

4. **How can you define new commands?**

   You can define new commands using the : (COLON) and ; (SEMI-COLON) commands. These commands mark the beginning and end of a command’s definition.

5. **How can you read and write memory?**

   You can read from a memory address using the read, read.halfword, and read.byte commands. You can write a value to memory using the write, write.halfword, and write.byte commands.

6. **How can you give descriptive names to important numbers used in your scripts?**

   You can give a name to constant numbers using the command constant *constant-name*.

7. **How can you reserve a memory location to store a value?**

   You can reserve and name a memory location using the command value *variable-name*.

8. **How can you repeat the execution of a command sequence?**

   You can use the begin and again commands together to mark the beginning and end of a command sequence you want to repeat endlessly. You can use the do and loop commands together to mark the beginning and end of command sequences that you want to repeat a limited number of times.

9. **How can you conditionally execute command sequences?**

   You can create branches using the if and then commands. You use these commands together to mark the beginning and end of a conditional command sequence that executes when a condition has been met.

10. **How can you simplify management of intermediate values needed in your scripts?**

    You can use *local variables* in your commands to simplify managing the intermediate values in your commands' computations. You can define both *initialized local variables* and *uninitialized local variables* using the commands { (LEFT-CURLY-BRACE), | (PIPE), and } (RIGHT-CURLY-BRACE).

11. **How can you create your own custom test procedures?**

    You can create custom diagnostic tests using the test.begin, test.while, and test.repeat commands. The test.begin and test.repeat commands mark the beginning and end of your test procedure. The test.while command conditionally exits the test procedure